# REPORT DOCUMENTATION PAGE

*Form Approved*
OMB No. 0704-0188



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT DATE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| 31-07-2003 | Annual Technical Progress | 01 July 2002 - 30 June 2003 |

**4. TITLE AND SUBTITLE**
Language-based Security for Malicious Mobile Code

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**
N00014-01-1-0968

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Fred B. Schneider,
Dexter Kozen,
Greg Morrisett and
Andrew Myers

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Cornell University
Ithaca, NY 14853

**8. PERFORMING ORGANIZATION REPORT NUMBER**
39545

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Office of Naval Research
Ballston Centre Tower ONE
800 North Quincy Street
Arlington, VA 22217-5660

**10. SPONSOR/MONITOR'S ACRONYM(S)**
ONR

**11. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for Public Release; distribution is Unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
Report summarizes progress over the past year in developing language-based technologies for defending software systems against attacks from mobile code and system extensions.

20030812 086

**15. SUBJECT TERMS**
In-lined reference monitors, proof carrying code, end-to-end security, information flow enforcement.

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT<br>U | b. ABSTRACT<br>U | c. THIS PAGE<br>U | UU | | 19b. TELEPONE NUMBER (*Include area code*) |

# Language-based Security for Malicious Mobile Code

N00014-01-1-0968

Annual Report
01 July 2002 – 30 June 2003

Fred B. Schneider (Principal Investigator)
Dexter Kozen, Greg Morrisett, and Andrew Myers (Co-Investigators)

Department of Computer Science
Cornell University
Ithaca, New York 14853

## Overview

This project is investigating programming language technology—program analysis and program rewriting—for defending software systems against attacks from mobile code and system extensions. The approach promises to support a wide range of flexible, fine-grained access-control and information-flow policies. Only a small trusted computing base seems to be required. And the run-time costs of enforcement should be low.

Our progress over the past year is summarized below. Details can be found in the publications whose citations are given following all the summaries. A list of DoD interactions and technology transitions appears at the end of the report.

## In-lined Reference Monitors

This past year, working with Ph.D. student Kevin Hamlen, Morrisett and Schneider developed a more refined characterization of what policies can be enforced using reference monitors. This new work extends earlier work by Schneider, now taking into account the limits of computability. Specifically, we developed a model based on standard Turing machines, adapted Schneider's criteria for enforceable security policies, and introduced computability

requirements. We also integrated static analysis and program rewriting into the model.

By providing this unifying model, and by basing it on Turing machines, we were able to compare the relative power of the various enforcement mechanisms, and to relate them to standard computability results. For instance, it was relatively easy to show that the class of policies precisely supported by static analysis could also be supported by both reference monitors and by program rewriting. In addition, we found that introducing a computability requirement on reference monitors was necessary, but not sufficient, for precise characterization of the class of policies actually realizable by reference monitors. And we identified a new property, which we call "punctuality" that provides a more accurate upper bound on the power of reference monitors.

Our most surprising and important results involve program rewriting. We can show that the class of policies originally characterized by Schneider does not include all policies enforceable through rewriting (and vice versa). Indeed, we were able to show that the class of policies enforceable through rewriting does not correspond to any class of the Kleene hierarchy. This is a surprising and important result, as it shows that rewriting truly is a powerful security enforcement technique.

**Progress on Prototype IRM.** Last year, we developed a prototype IRM rewriter for the Microsoft CIL, which takes a limited class of policies written in a very primitive specification language. In essence, the policy writer could only specify that certain (non-virtual) method calls should be replaced with alternative method calls. Though limited, we showed that this tool could be used to effectively enforce practical policies.

This year, we have extended the rewriting tools so that we can perform arbitrary rewriting on the CIL code. This was accomplished by building on a bytecode-rewriting toolkit developed by Microsoft Researchers. In Fall 2002, Kevin Hamlen and Greg Morrisett visited Microsoft Research in Cambridge to further develop the APIs and code for doing this manipulation.

## Cyclone Compiler

Today, our computing and communications infrastructure is built using unsafe, error-prone languages such as C or C++ where buffer overruns, format string errors, and space leaks are not only possible, but frighteningly common. In contrast, type-safe languages, such as Java, Scheme, and ML,

ensure that such errors either cannot happen (through static type-checking and automatic memory management) or at least are caught at the point of failure (through dynamic type and bound checks). This fail-stop guarantee is not a total solution, but it does isolate the effects of failures, facilitates testing and determination of the true source of failures, and it enables tools and methodologies for achieving greater levels of assurance.

The obvious question is: "Why don't we re-code our infrastructure using type-safe languages?" Though such a technical solution looks good on paper, the cost is simply too large. For instance, today's operating systems consist of tens of millions of lines of code. Throwing away all of that C code and reimplementing it in, say Java, is simply too expensive.

As a step towards these goals, we have been developing Cyclone, a type-safe programming language based on C. The type system of Cyclone accepts many C functions without change and uses the same data representations and calling conventions as C for a given type constructor. The Cyclone type system also rejects many C programs to ensure safety. For instance, it rejects programs that perform (potentially) unsafe casts, that use unions of incompatible types, that (might) fail to initialize a location before using it, that use certain forms of pointer arithmetic, or that attempt to do certain forms of memory management.

All of the analyses used by Cyclone are local (i.e., intra-procedural) so that we can ensure scalability and separate compilation. The analyses have also been carefully constructed to avoid unsoundness in the presence of threads. The price paid is that programmers must sometimes change type definitions or prototypes of functions, and occasionally they must rewrite code.

We find that programmers must touch about 10% of the code when porting from C to Cyclone. Most of the changes involve choosing pointer representations and only a very few involve region annotations (around 0.6 % of the total changes). This past year, we developed a semi-automatic tool that can be used to automate most of these changes.

The performance overhead of the dynamic checks depends upon the application. For systems applications, such as a simple web server, we see no overhead at all. This is not surprising, as these applications tend to be I/O-bound. For scientific applications, we were seeing a much larger overhead (around 5x for a naive port, and 3x with an experienced programmer), due to array bounds and null pointer checks. To avoid these, over the past year we incorporated a sophisticated intra-procedural analysis that eliminates most of those checks. For instance, a simple matrix-multiply now runs as fast as C code, where before, it was taking over 5x as long.

We also introduced new typing mechanisms that support a wider range of safe memory management options. Before, we had to restrict programmers to using only garbage collection, stack allocation, or limited forms of region allocation, all of which could adversely affect time and space requirements. This year, we added support for dynamic region allocation, unique pointers, and reference-counted objects. These mechanisms let programmers control memory management overheads without sacrificing safety. For instance, we were able to improve the throughput of the MediaNet streaming media server by up to 42% and decrease the memory requirements from 8MB to a few kilobytes using these new features.

Finally, as part of his Ph.D. dissertation, Daniel Grossman designed extensions that support type-safe multi-threading. These extensions, which we plan to implement in the next year, statically ensure the absence of data races in programs, thereby avoiding another wide class of security problems.

## Secure Program Partitioning

We continue our work in developing *secure program partitioning*, a novel way to ensure that data confidentiality and integrity are preserved in distributed systems that contain untrusted hosts and mutually distrusting principals. This problem is particularly relevant to information systems used by mutually distrusting organizations, such as the dynamic coalitions that arise in military settings.

In our approach, programs are automatically partitioned into communicating subprograms that run on the available, partially trusted hosts. The partitioning automatically extracts a secure communications protocol, so that if any host is subverted, then only those principals that have explicitly stated trust in that host need fear a violation of confidentiality. That is, for a given principal $p$, the partitioned program we create is robust against attacks on hosts not trusted by $p$. To protect data integrity, information and code are also replicated across the available hosts. Some replicas may be securely hashed to protect them against subversion of the host on which they are executing.

We have implemented these techniques in Jif/split, an extension to our publicly released Jif compiler that statically enforces information flow control, in conjunction with a distributed run-time system that securely executes partitioned and replicated programs while guarding against subverted or malicious hosts. New protocols have been developed to permit secure transfer of control between one group of host replicas and another. To un-

derstand the practicality of our approach, secure distributed systems have been implemented using Jif/split, including various secure auction protocols. Performance of the system is quite reasonable, despite the fine-grained program partitioning. We are now investigating availability policies in this framework, which should help defend against denial of service attacks.

**Information Flow Semantics.** We have also been investigating how to define and enforce information flow policies in concurrent, probabilistic, and nondeterministic systems. Concurrent systems are naturally nondeterministic, because a thread scheduler must decide when to allow various threads to execute, and this decision is beyond the programmer's control. Nondeterminism is dangerous, because it allows covert communication between threads, using timing. We have given a new formal definition of security for concurrent systems; this definition seems to correspond more closely to an intuitive notion of security than previous definitions do. Further, we have defined an expressive core concurrent programming language, which is equipped with a type system for a static analysis that ensures programs written in this language are secure. Incorporation of this static analysis into the Jif framework is an obvious next step.

In related ongoing work, we are exploring expressive security conditions for systems incorporating probabilistic and nondeterministic computation. Our goal is to bound information flows. Most information flow analyses are useful only for showing that there is no information flow, but many real-world systems (for example, password checkers) leak acceptable amounts of information. We have developed an appropriate program semantics for modeling such systems and are working towards a logic that can be used to prove expressive assertions about bounded information flow.

## Avoiding Malicious Firmware

After power-up, most computing devices enter a *boot phase* in which the hardware configuration is recognized, devices are initialized, and the operating system is loaded and started. The program that controls this process is called *boot firmware* and is typically stored in ROM or other non-volatile memory. The recent trend is that boot firmware is becoming more complex, as it gains additional responsibilities. Until recently, malicious boot firmware received relatively little attention. Several factors now conspire to make for a very worrisome form of attack:

- Boot firmware runs in a privileged mode on bare hardware, prior to

the start of most security services. Malicious boot firmware could cause harm in several different ways, most notably by corrupting the operating system. Many current security mechanisms assume that the operating system can be trusted. Thus, malicious boot firmware could subvert most of the security mechanisms currently deployed at the OS, application, and enterprise levels.

- Modern boot firmware often consists of modules contributed by multiple vendors, many of whom might not be visible to end users. Often these modules are boot-time device drivers for distinct pieces of hardware. The configuration of boot firmware can be quite volatile as the hardware configuration changes. Many devices support semi-automated firmware upgrades, so there are many opportunities to introduce malicious boot firmware.

Thus, we consider malicious boot firmware to be a plausible, practical, and dangerous form of attack. Exploiting this vulnerability should be well within the means of motivated adversaries such as nation-states and criminal organizations.

We are focused on detection of malicious boot firmware within systems based on Open Firmware. Open Firmware is a mature and widely used standard for boot firmware. Sun Microsystems and Apple both use boot firmware that conforms to the standard. The most salient feature of Open Firmware is that it includes an interpreter (or virtual machine) for fcode, a lightly compiled form of the Forth programming language.

Fcode device drivers, supplied by a wide range of relatively anonymous vendors, pose a significant risk of introducing malicious code into the boot program. Our concern is with detection of malicious fcode using static checks performed during each boot cycle. We check potentially dangerous, untrusted code each time, prior to execution.

Our safety policy is baked-in; there is no need for the user to specify anything. The policy consists of three tiers.

**Tier 1: Basic Safety.** Included in this tier are type safety, memory safety, stack safety, and control-flow safety. Type safety is the requirement that each storage location and each computational result has a well-defined type that can be determined by static analysis prior to running the program. All assignments and memory references must respect those types. Memory safety is the requirement that all memory accesses are to legal (i.e., allocated) locations. Stack safety is the requirement that the program obeys

an appropriate discipline with respect to its own calling stack. Control-flow safety is the requirement that jump targets are locations containing executable instructions within an appropriate subprogram.

**Tier 2: Device Encapsulation.** Code supplied by different vendors, typically device drivers, will be loaded into the boot program and must coexist. We require that these programs respect each others' boundaries, and they only interact through published interfaces. Of critical importance is the requirement that each device be operated solely by its own device driver.

**Tier 3: Structural Safety.** Code supplied by vendors will interact with Open Firmware services through an API that prevents unsafe calls. This safety arises for three reasons: the API exposes only a restricted, safer subset of functionality, the implementation performs runtime checks, and our verifier can further restrict the way the API is invoked. What must be verified is that the untrusted program really uses the API as specified and does not bypass it or tamper with its implementation.

For programs written in high-level languages, these properties are clear and often implicit in the definition of the language. For instance, Java enforces almost all type correctness at compile time. Language features, such as Java's private modifier, can be used to enforce modularity. In such languages, interaction between specific code modules is evident on inspection. However, our verification is performed on fcode, a primitive language in which none of this would be easy. Our verifier relies on the fact that the fcode program is the result of compiling from a high level language—Java. This special compiler produces particularly well-structured and annotated fcode, in which constructs derived from Java are readily recognized.

Our prototype system consists of three interlinked elements: the Java VM-to-fcode compiler J2F, the BootSafe verifier, and Java API for BootSafe-compliant Open Firmware drivers along with runtime support for the API. We are building prototypes of these three elements. Vendors will write device drivers in Java and use our compiler to generate fcode. Users will trust our verifier and our runtime support (both installed in their boot platform), but will not need to trust the device driver code received from the vendor.

## Publications Supported under this Grant


(1) Frank Adelstein, Dexter Kozen, and Matt Stillerman. Malicious code detection for open firmware. *Proc. 18th Computer Security Applications Conf. (ACSAC'02)*, Dec. 2002, 403–412.

(2) James Cheney and Ralf Hinze. Poor man's generics and dynamics. *Haskell Workshop 2002* (Pittsburgh, PA, September 2002).

(3) James Cheney and Christian Urban. System description: Alpha-Prolog, a fresh approach to logic programming modulo alpha-equivalence. *Workshop on Unification* (Valencia, Spain, May, 2003).

(4) Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567-600, November 2002.

(5) Robert Givan, David McAllester, Carl Witty, and Dexter Kozen. Tarskian set constraints. *Information and Computation 174*, No. 2 (2002), 105–131.

(6) Daniel J. Grossman. *Safe Programming at the C Level of Abstraction.* Ph.D. Thesis, Cornell University, August 2003.

(7) Daniel J. Grossman. Type-Safe Multithreading in Cyclone. *ACM Workshop on Types in Language Design and Implementation* (New Orleans, LA, January 2003).

(8) David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 4, Kluwer, 2nd edition, 2002, 99–217.

(9) Dag Johansen, Robbert van Renesse, and Fred B. Schneider. WAIF: Web of Asynchronous Information Filters. *Future Directions in Distributed Computing*, Lecture Notes in Computer Science, Volume 2585 (Schiper, Shvartsman, Weatherspoon, and Zhao, eds.) Springer-Verlag, 2003, 81–86.

(10) Dexter Kozen. On the complexity of reasoning in Kleene algebra. *Information and Computation 179* (2002), 152–162.

(11) Dexter Kozen. On Hoare logic, Kleene algebra, and types. In P. Gärdenfors, J.Woleński, and K. Kijania-Placek, editors, *In the Scope*

*of Logic, Methodology, and Philosophy of Science*, Volume 1 of the 11th Int. Congress Logic, Methodology and Philosophy of Science, (Cracow, August 1999), volume 315 of Studies in Epistemology, Logic, Methodology, and Philosophy of Science, Kluwer (2002), 119–133.

(12) Dexter Kozen. On two letters versus three. In Zoltán Ésik and Anna Ingólfsdóttir, editors, *Proc. Workshop on Fixed Points in Computer Science (FICS'02)*, July 2002, 44–50.

(13) Dexter Kozen. Some results in dynamic model theory (abstract). In E. A. Boiten and B. Möller, editors, *Proc. Conf. Mathematics of Program Construction (MPC'02)*, Lecture Notes in Computer Science, Volume 2386, Springer-Verlag, July 2002, 21.

(14) Dexter Kozen and Matt Stillerman. Eager class initialization for Java. In W. Damm and E.R. Olderog, editors, *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, Lecture Notes in Computer Science, Volume 2469, Springer-Verlag, Sept. 2002, 71–80.

(15) Dexter Kozen and Jerzy Tiuryn. Substructural logic and partial correctness. *Trans. Computational Logic 4*, No. 3 (2003), 355–378.

(16) Jed Liu and Andrew C. Myers. JMatch: Iterable abstract pattern matching for Java. *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages* (New Orleans, LA, January 2003), 110–127.

(17) Yaron Minsky and Fred B. Schneider. Tolerating malicious gossip. *Distributed Computing 16*, 1 (Feb 2003), 49–68.

(18) Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. *Proceedings of the 12th International Conference on Compiler Construction* (Warsaw, Poland, April 2003), Lecture Notes in Computer Science, Volume 2622, 138–152.

(19) Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21*, No. 1 (January 2003), 5–19.

(20) Fred B. Schneider. Least privilege and more. *Computer Systems: Papers for Roger Needham*, Andrew Herbert and Karen Sparck Jones, eds. Microsoft Research, 2003, 209–213.

(21) Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *Journal of Functional Programming*, 13(3):677-708, May 2003.

(22) Matt Stillerman and Dexter Kozen. Demonstration: Efficient code certification for open firmware. *Proc. 3rd DARPA Information Survivability Conference and Exposition (DISCEX III)*, volume 2, IEEE Computer Society, Los Alamitos, CA, April 2003, 147–148.

(23) Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computing Systems, 20*, No. 3 (August 2002), 283–328.

(24) Stephan A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.

(25) Steve Zdancewic and Andrew C. Myers. Secure information flow and linear continuations. *Higher Order and Symbolic Computation 15*, Nos. 2–3 (Sept. 2002), 209–234.

(26) Steve Zdancewic and Andrew C. Myers. Secure information flow and linear continuations. *Higher Order and Symbolic Computation 15*, Nos. 2–3 (Sept. 2002), 209–234.

(27) Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. *Proceedings of the 16th IEEE Computer Security Foundations Workshop* (Pacific Grove, California, June 2003), 29–43.

(28) Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, May 2003), 236–250.

## DoD Interactions and Technology Transitions

- As a consultant to DARPA/IPTO, Schneider chairs the independent evaluation team for the OASIS Dem/Val prototype project. This project funds two consortia to design a battlespace information system intended to tolerate a class A Red Team attack for 12 hours.

- Greg Morrisett spent nine months visiting Microsoft's Cambridge Research Laboratory, where he worked with researchers on programming language and security technology. In particular, Morrisett worked on the development of Microsoft's tools for automatically finding security flaws in production code, based on his experience with Cyclone. He also worked with student Kevin Hamlen and Microsoft researchers on the implementation of the .NET rewriting tool for inline reference monitors.

- Further public releases of Myers' Jif compiler have been made available at the Jif web site, http://www.cs.cornell.edu/jif. The Jif language extends the Java programming language with support for information flow control. The Jif compiler is implemented on top of the Polyglot extensible compiler framework for Java. The Polyglot framework has also been released publicly at http://www.cs.cornell.edu/projects/polyglot, and researchers at Princeton University are using this framework in their own research. The releases of both Jif and Polyglot are provided as Java source code and work on Unix and Windows platforms.

- AT&T research is working with us to develop the Cyclone language, compiler, and tools. In addition, researchers at the University of Maryland, the University of Utah, Princeton, and the University of Pennsylvania, and Cornell are all using Cyclone to develop research prototypes.